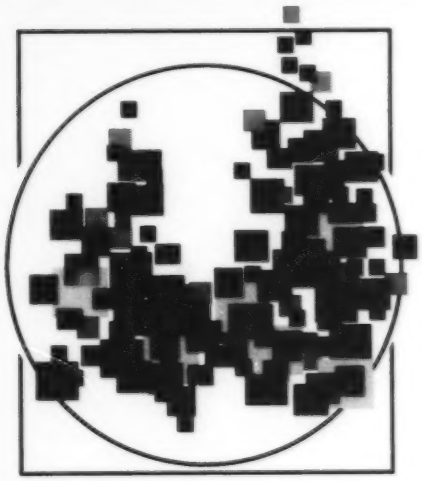

CENTRE FOR
SYSTEMS
SCIENCE

Simon Fraser University, Burnaby, B.C., Canada V5A 1S6 / (604) 291-4704

92-05399/

The HPSG-PL System

Fred Popowich and Carl Vogel

CSS-IS TR 91-08
CMPT TR 91-08

October 1991

The HPSG-PL System

Fred Popowich
Carl Vogel

CMPT TR 91-08
CSS-IS TR 91-08

School of Computing Science
Centre for Systems Science
Simon Fraser University
Burnaby, B.C.
Canada V5A 1S6

October 1991

Abstract

HPSG-PL is a Prolog implementation of Head Driven Phrase Structure Grammar. The system consists of a lexical compiler, constraint processor, chart parser and a module for linking the parser to a graphic interface. Using this system, a user can examine the properties of the HPSG formalism itself, and can investigate characteristics of specific grammars that utilize the formalism.

Contents

1 Introduction 1

2 The Lexicon 1

3 Principles and Grammar Rules 6

4 Running the System 9

4.1 Starting the System 9

4.2 Loading a Grammar 9

4.3 Parsing Sentences 10

4.4 Debugging a Grammar 11

4.5 Error Messages 12

5 Summary 13

A Sample Grammar 14

B Terminal Session 21

Preface

This document describes version 1.1 of HPSG-PL. It differs from version 1.0 in that it allows "lexical entries" for unknown words. The internal format for storing lexical entries has been modified slightly. The user interface has been improved to allow ".typ" and ".lex" files to be loaded separately. Additional debugging predicates have also been included.

The implementation has been influenced by a range of other grammar development systems and parsers [EG84, Pop85, MCK+89]. HPSG-PL can be used for noncommercial purposes provided the authors are appropriately acknowledged. The system may be obtained by: 1) writing to Fred Popowich, School of Computing Science, Simon Fraser University, Burnaby, B.C., V5A 1S6, or 2) sending electronic mail to popowich@cs.sfu.ca.

1 Introduction

Head Driven Phrase Structure Grammar (HPSG) [PS87] is a *lexical, unification based* grammar formalism for describing the structure and meaning of natural language sentences. It is classified as *lexical* since it has very few grammar rules and thus relies to a high degree on the lexicon. The grammar introduced in [PS87], for example, contains only four grammar rules. The lexicon in HPSG is more than just a dictionary, it is a highly structured knowledge base which uses *lexical types* and *lexical rules* to reduce redundancy and capture linguistic generalizations. HPSG is described as *unification-based* since it uses unification as its basic operation for combining the linguistic information associated with different words, rules and principles.

As the name suggests, HPSG-PL is a Prolog implementation of the HPSG formalism, which is based on the implementation described in [PV91]. It makes use of a chart parser [PV90] and has access to various other modules including a graphic module [BHP90]. The implementation closely follows the version of HPSG described in [PS87], except that it does not incorporate lexical rules or morphological analysis.

When developing a natural language interface for a database or an expert system, one needs to determine the different classes of words and sentence constructions that will be allowed. This requires the specification of dictionary entries and grammar rules. Using a lexical formalism like HPSG, the process of grammar design is reduced to one of lexicon design; little or no modification is needed to the grammar rules in the design of grammars for different applications.

2 The Lexicon

Within HPSG, all information is organized through the use of *signs*. The signs themselves as represented as attribute-value matrices. An attribute value matrix (AVM) is a collection of attribute value pairs (AVPs), where the attributes are atomic, and the values may be either atoms, AVMs, lists or sets.

The sign introduced in (1) corresponds to the lexical entry for the finite verb *walks*. AVMs are enclosed in square brackets, while lists are enclosed in angle brackets, with sets being enclosed in braces.

$$\left[\begin{array}{ll} \text{PHON} & \langle \textit{walks} \rangle \\ \text{SYN} & \left[\begin{array}{ll} \text{LOC} & \left[\begin{array}{ll} \text{HEAD} & \left[\begin{array}{l} \text{MAJ v} \\ \text{FORM fin} \\ \text{INV} - \\ \text{AGR } \boxed{1} \left\{ \left[\begin{array}{l} \text{PER 3} \\ \text{NUM sing} \end{array}\right] \right\} \end{array}\right] \\ \text{LEX} & + \\ \text{SUBCAT} & \langle \text{NP[nom,}\boxed{1}\text{]} \rangle \end{array}\right] \end{array}\right] \end{array}\right] \quad (1)$$

The sign contains a top-level attribute for phonological information (PHON) which has

as its value a list containing the orthographic representation of ***walks***. SYN contains the syntactic information associated with the lexical entry. The value of this attribute is an AVM containing local (LOC), lexical (LEX) and subcategorization (SUBCAT) features. Observe that the LEX feature takes an atomic value, while the SUBCAT feature takes a list containing the element *NP[nom, [1]]* which is an abbreviation for the sign shown below.

$$\text{NP[nom,}\boxed{1}\text{]} = \left[\text{SYN}\left[\text{LOC}\left[\begin{array}{l}\text{HEAD}\left[\begin{array}{l}\text{MAJ n}\\ \text{CASE nom}\\ \text{AGR }\boxed{1}\end{array}\right]\\ \text{SUBCAT}\ \langle\ \rangle\end{array}\right]\right]\right] \tag{2}$$

The agreement attribute (AGR) within the HEAD feature of LOC is a set valued feature containing a single AVM for third person singular agreement. This value is preceded by the index [1] to illustrate that this same set is the value of the AGR feature of the noun within the SUBCAT feature—indices like [1] indicate structure sharing, or re-entrancy.

Since the signs used in an HPSG grammar contain a great deal of information, the formalism introduces a hierarchy of *lexical types* in order to capture linguistic generalizations and to reduce redundancy in the lexicon. For example, we can define the lexical type for intransitive verbs to be a sign that contains the information common to all intransitive verbs.

$$\textit{intransv} \equiv_{\Delta} \left[\text{SYN}\left[\text{LOC}\left[\begin{array}{l}\text{HEAD}\left[\begin{array}{l}\text{MAJ v}\\ \text{FORM fin}\\ \text{INV} -\\ \text{AGR }\boxed{1}\{\ldots\}\end{array}\right]\\ \text{LEX} +\\ \text{SUBCAT}\ \langle \text{NP[nom,}\boxed{1}\text{]}\rangle\end{array}\right]\right]\right] \tag{3}$$

Similarly,

$$\textit{3rdsing} \equiv_{\Delta} \left[\text{SYN}\left[\text{LOC}\left[\text{HEAD}\left[\text{AGR}\left\{\left[\begin{array}{l}\text{PER 3}\\ \text{NUM sing}\end{array}\right]\right\}\right]\right]\right]\right] \tag{4}$$

Then the lexical entry for ***walks*** can be defined in terms of these two lexical types.

$$_{\textit{intransv}\ \wedge\ \textit{3rdsing}}\left[\text{PHON}\ \langle \textit{walks}\rangle\right] \tag{5}$$

The abbreviation *NP[nom, [1]]* that we introduced in (2) can be viewed as a parameterized lexical type (6), where X and Y are parameters.

$$\textit{np}[X,Y] \equiv_{\Delta} \left[\text{SYN}\left[\text{LOC}\left[\begin{array}{l}\text{HEAD}\left[\begin{array}{l}\text{MAJ n}\\ \text{CASE X}\\ \text{AGR Y}\end{array}\right]\\ \text{SUBCAT}\ \langle\ \rangle\end{array}\right]\right]\right] \tag{6}$$

HPSG-PL converts symbolic representations of signs into Prolog lists (of a fixed arity) which can then be combined using Prolog's term unification. Signs are implemented as lists

of AVPs. Each AVP is implemented as a two element list, where the first element is an atom corresponding to the attribute, and the second is either an atom, list or an AVM. We shall defer our discussion of the implementation of set valued features until later.

The structure of signs is described via *attribute value (AV) declarations*[1]. AV declarations are required only for those attributes which take AVMs as their values. The declarations describe the structure of an AVP that makes up an AVM as shown below.

$$A \texttt{ has } [B_1, B_2, \ldots B_n]. \tag{7}$$

where A is an attribute name and $[B_1, B_2, \ldots B_n]$ is the structure of the AVM that is the value of A. Each B_i will correspond to an AVP as well, consisting of an attribute A_i and some value. If A_i is a list, set or atomic valued-attribute, then we specify B_i as $!A_i$. This is equivalent to saying that there is an AVP consisting of attribute A_i and some value *whose structure we do not care about at this time!* For attributes that take AVMs as their values, we will usually want to describe the structure of these values. In this case, the B_i can reference a *previously* defined AVP for an attribute A_i through the specification $@A_i$. For example, to define the sign structure that we have been using in our examples, we would need the following declarations.

```
head    has  [!inv, !agr, !form, !case].
loc     has  [@head, !subcat, !lex].
syn     has  [@loc].
```

The first declaration states that the head feature has as its value an AVM containing the features inv, agr, form and case. For each of the three subfeatures, the values are declared to be 'unspecified.' This declaration constructs a structure of the form

```
[head, [ [inv,_], [agr,_], [form,_], [case,_] ] ]
```

The second declaration describes the structure of the loc feature and accesses the definition of the head feature shown earlier. The final declaration describes the syn feature.

The order of the declarations is important. For instance head must be declared before its structure is referenced with @head. Recursive or forward references are not allowed. Consequently, the following declarations where @syn is used in the declaration for slash is not allowed.

```
slash    has  [@syn, @sem].
non_loc  has  [@slash].
syn      has  [@loc, @non_loc].
```

We can describe a structure in which a syn attribute appears within a slash attribute which is itself within a syn attribute if we do the following.

```
slash    has  [!syn, @sem].
non_loc  has  [@slash].
syn      has  [@loc, @non_loc].
```

[1]These were called AV types in [PV91].

Here we declare that slash has a syn feature but we do not declare any internal structure of this feature (!syn). Alternatively, we could give two definitions for the structure of a syn feature, one of which has a less specified value as illustrated below.

```
syn-1    has  [@loc].
slash    has  [@syn-1, @sem].
non_loc  has  [@slash].
syn-2    has  [@loc, @non_loc].
```

Observe the use of the suffixes -1 and -2 to distinguish between the two definitions for syn. The suffix is *not* incorporated into the attribute name as is illustrated in the following expansion of the syn-1 declaration which contains syn and not syn-1 as the attribute name.

```
[syn, [
   [loc, [
      [head, [
         [inv,_],
         [aux,_],
         [agr,[[num,_],[per,_],[gen,_]]],
         [form,_],
         [case,_],
         [adjuncts,_]]],
      [subcat,_],
      [lex,_]]]]]
```

Currently, the only *set-valued* features that are allowed by the system are sets of atomic values—sets of AVMs are not allowed. The domain of a set valued feature must first be declared with a domain statement of the form

$$S \texttt{ domain } [V_1, V_2, \ldots V_n]. \tag{8}$$

where S is the name of the set, and the V_i are all the possible elements of a set. For example, we can declare the set of cases for noun phrases as shown below.

```
case   domain  [nom, acc, gen, dat].
```

The set command can then be used to describe a set of values from S

$$S \texttt{ set } [X_1, X_2, \ldots X_m]. \tag{9}$$

where $m \leq n$, and each X_j corresponds to a distinct V_i. For example, the set {nom, acc} would be represented as

```
case   set  [nom, acc].
```

The value of the unification of two sets is defined to be their intersection. If unification results in the empty set then unification fails. Sets are implemented as terms using a technique first developed by Alain Colmerauer which is described by Mellish in [Mel88]. A set having a domain of n elements will be converted into a term with $n+1$ arguments, and the unification of sets can be obtained directly with term unification.

Just as the **has** relation is used to define macros for AVPs, the **type** relation is used to define macros for AVMs. These macros correspond to the lexical types that we described earlier in this section. The definition of a lexical type can contain references to previously defined lexical types. It may also contain *path equations* for accessing values associated with attributes that have been declared through the AV declarations. The form of an AV type definition for a type *Name* is

$$\textit{Name}\ \texttt{type}\ \ M_1\ \&\ M_2\ \&\ \ldots\ \&\ M_m\ \texttt{with}\ P_1\ \texttt{with}\ P_2\ \ldots\ \texttt{with}\ P_n\,. \tag{10}$$

where $m \geq 1$ and $n \geq 0$. *Name* may be an arbitrary Prolog term, while each M_i may be either a list of references to previously defined AVPs, or a reference to a previously defined lexical type. The AVMs described by each M_i are unified to obtain the AVM described by *Name*. Each path equation P_j is of the form

$$A_1{:}A_2{:}\ldots{:}A_p\ =\ V$$

V may be an atom, a variable, a function call, a list, an AVM or even a reference to a lexical type. Consider the following example.

```
sign    type [!dtrs, !phon, @syn, !sem].

lsign   type @sign with dtrs = [].

psign   type @sign with
   dtrs = [!head_dtr, !comp_dtrs] with
   dtrs:head_dtr = @sign.
```

Observe that in the definition of **sign** above, we did not spell out the structure of the **dtrs** attribute—as reflected by the use of **!dtrs**. The definition for lexical signs (**lsign**) uses a path equation to specify the value of the **dtrs** feature as the empty list. For phrasal signs (**psign**), the **dtrs** feature is given a list of AVPs (an AVM) as its value. The head daughter is then specified to have a **@sign** as its value. The order of the path equations in the **psign** definition is important. If we tried to access the **dtrs:head_dtr** attribute before we had specified the structure of the **dtrs** attribute using **dtrs = [!head_dtr, !comp_dtrs]** , then we would have received an error message.

The current version has the restriction that the name of an AV declaration cannot be the same as the name of an AV type. This restriction will be removed in the next version.

Lexical types and AV types are stored in clauses of the form

$$\texttt{sign}(\textit{Name}, \textit{Structure}, \textit{Constraints})\,.$$

as shown in the internal format for the **psign** type below.

```
sign(
  psign,
  [  [dtrs, [
        [head_dtr,[
           [dtrs,_],
           [phon,_],
           [syn,...],
           [sem,_]]],
        [comp_dtrs,_]]],
     [phon,_],
     [syn,...],
     [sem,_]],
  [])
```

We have not shown the internal structure of the **syn** feature since it was shown earlier.

The structure of lexical entries is similar to that of lexical types, except that the keyword **type** is replaced by the keyword **entry**. The *Name* of a lexical entry is the word that the lexical entry describes. If the *Name* is omitted, then the value of the **phon** attribute is used. A possible lexical entry for the intransitive verb ***walks***, which makes use of the lexical types *intransv* and *3rdsng* is portrayed below.

```
walks entry @intransv & @3rdsign with
   phon = [walks].
```

3 Principles and Grammar Rules

The principles and grammar rules of HPSG are stated in terms of signs. They describe relationships between the PHON, SYN and SEM information of a phrasal sign and that contained in its daughter signs.

Principles are stated in terms of the relative pseudocomplement operator $\Rightarrow$ as shown below, where X is a sign that various from principle to principle.

$$[\text{DTRS}_{\text{headed-structure}}[]] \Rightarrow X$$

A simplified view of the $\Rightarrow$ operator is that it ensures that all of the principles apply to each of the grammar rules. HPSG possesses a Constituent Ordering Principle, Head-Feature Principle, Subcategorization Principle, Semantics Principle, Adjuncts Principle, and Binding Inheritance Principle. For example, the Constituent Order Principle simply states that the phonology of a phrasal sign is determined by applying the *order* function to its daughters.

$$\left[\text{DTRS}_{\text{headed-structure}}\left[\ \right]\ \right] \Rightarrow \left[\begin{array}{l}\text{PHON order}(\boxed{1})\\ \\ \text{DTRS } \boxed{1}\end{array}\right] \tag{11}$$

The definition of the *order* function must incorporate the linear precedence constraints for the language in question. For English, lexical head daughters precede all their complements

in a sentence, while nonlexical head daughters follow all of their complements. The relative ordering of the complements is determined by their position in the subcategorization list of the head daughter.

Grammar rules of HPSG are Immediate Dominance Rules—they describe immediate constituency of constituents without imposing any restriction on their relative ordering. There are separate constraints, implicit within the Constituent Ordering Principle, which determine the order of sister constituents. Grammar rules are simply phrasal signs which impose constraints on the different daughters. Consider the following grammar rule from [PS87].

$$\begin{bmatrix} \text{SYN|LOC|SUBCAT} \left\langle\ \right\rangle \\ \\ \text{DTRS} \begin{bmatrix} \text{HEAD-DTR|SYN|LOC|LEX} - \\ \text{COMP-DTRS} \left\langle [\ ] \right\rangle \end{bmatrix} \end{bmatrix} \tag{12}$$

When taken in conjunction with the Constituent Order Principle, Head Feature Principle and Subcategorization Principle, it is equivalent to the following.

$$\begin{bmatrix} \text{PHON } \textit{order}(\boxed{3}) \\ \text{SYN|LOC} \begin{bmatrix} \text{HEAD } \boxed{1} \\ \text{LEX} - \\ \text{SUBCAT} \left\langle\ \right\rangle \end{bmatrix} \\ \text{DTRS } \boxed{3} \begin{bmatrix} \text{HEAD-DTR|SYN|LOC} \begin{bmatrix} \text{HEAD } \boxed{1} \\ \text{LEX} - \\ \text{SUBCAT } \boxed{2} \left\langle [\ ] \right\rangle \end{bmatrix} \\ \\ \text{COMP-DTRS } \boxed{2} \end{bmatrix} \end{bmatrix} \tag{13}$$

We view the unification of all the principles as defining the lexical type *principled*. Adopting this approach, the actual grammar rule can then be defined in terms of *principled*.

$$\text{principled} \begin{bmatrix} \text{SYN|LOC|SUBCAT} \left\langle\ \right\rangle \\ \\ \text{DTRS} \begin{bmatrix} \text{HEAD-DTR|SYN|LOC|LEX} - \\ \text{COMP-DTRS} \left\langle [\ ] \right\rangle \end{bmatrix} \\ \\ \end{bmatrix} \tag{14}$$

To define the lexical type *principled*, we must first define the lexical types associated with each of the principles. For example, we will define the lexical type cop for the Constituent Order Principle.

```
cop  type @psign with
   dtrs = Dtrs with
   phon = order(Dtrs).
```

This will be converted into a Prolog term of the following form.

```
sign(
  cop,
  [  [dtrs, Dtrs],
     [phon, Z],
     [syn, ...],
     [sem, _]],
  [order(Dtrs,Z)])
```

Observe that the value of the phon attribute is specified to be Z, and that there is a constraint `order(Dtrs,Z)` which will be evaluated when the phonology attributes of the `Dtrs` becomes instantiated.[2] After providing similar definitions for the other principles, we can then define `principled`.

```
principled type @cop & @hfp & @subp & @semp & @adjp & @bip.
```

Currently, only five functions are recognized by the system. Their behaviour is summarized below.

order
: Given an AVM that appears as the value of a DTRS attribute, the `order` function returns a list containing the values of the PHON features of the individual daughters.

append
: Given two lists, the append function returns a single list containing the elements of the first list following by those of the second list.

select
: Given a list, the `select` function returns a list of *copies* of elements from the input list. This function is required by the grammnar rule for adjuncts.

select1
: An alternative form of the `select` function which when given a list, returns a copy of a single element from the list.

combine
: Given the semantic content of a head daughter, a list of complements, and a list of adjuncts, this function returns the AVM resulting from unifying the SEM|CONT of each adjunct and complement with that of the head daughter.

Additional functions can be defined by providing Prolog definitions for the appropriate predicates (in the `constraints.pl` file).

In the HPSG-PL system, the format for specifying grammar rules is similar to that for lexical entries. The keyword `rule` is used for grammar rules. The name of the grammar rule is useful for debugging and for tracing.

[2] As described in [PV91], function calls of the form $f(X,Y)$ are implemented in terms of a constraint $f(X,Y,Z)$ where Z is the value of the function $f(X,Y)$. Z is placed in the *Structure* as the value of the function call. Since the system was implemented under Quintus Prolog, a deferral mechanism had to be added so that functions would not be evaluated until their arguments were instantiated. Under Sicstus Prolog, the `freeze` predicate can be used for this purpose.

```
1    rule    @principled with
             dtrs:head_dtr:syn:loc:lex = minus with
             syn:loc:subcat = [] with
             dtrs:comp_dtrs = [_].
```

4 Running the System

The HPSG-PL system consists of a lexical compiler (`lex.pl`), a chart parser (`parser.pl`), a constraint processor (`constraints.pl`), and an interface to the graphic module TreeTool (`treetool.pl`). The lexical compiler converts the symbolic representations of lexical types, lexical entries and grammar rules described in the previous sections into Prolog clauses which can be used directly by the parser. The chart parser uses the constraint processor while considering all possible parses of a sentence. The constraint processor contains definitions for the various *functions* that may appear within the signs, plus it determines *when* to evaluate these functions. The TreeTool interface can be used to display the parse tree for successful parses, or it can be used to examine the various constituents contained in the chart.

4.1 Starting the System

The system may be started from Prolog by entering the goal

```
?- [main].
```

This will result the system automatically loading all of the necessary system files.

4.2 Loading a Grammar

A grammar consists of a collection of AV pair declarations, lexical types, grammar rules and lexical entries. We have adopted the convention whereby all of the grammar, with the exception of the lexical entries, is placed in a file with the suffix "`.typ`" with the lexical entries being placed in a file with the suffix "`.lex`". Entering the command

```
?- gram(test).
```

will clear any previously defined grammar and will invoke the lexical compiler on the files `test.typ` and `test.lex`. The command

```
?- typ(test).
```

will clear any previously defined declarations, lexical types and grammar rules, but it will only process the file `test.typ`. The lexicon `test.lex` can then be subsequently loaded with

```
?- lex(test).
```

Invoking the `lex` command will only clear previously defined lexical entries, it will not affect the lexical types, rules, etc.

4.3 Parsing Sentences

The parser is invoked by entering the goal

```
?- parse.
```

The user is then prompted for a sentence which must be terminated with a period. For each successful parse, the elapsed CPU time in displayed along with the value of the semantic attribute of the sign associated with the successful parse. After all alternatives have been exhausted, the total elapsed time is displayed along with the number of edges in the chart.

There is a collection of flags that affect the behaviour of the parser and the system output. The defaults for all these flags is off. They may be set by entering the goal

```
?- +flag.
```

where *flag* is one of `verbose`, `phrasal`, `explicit`, or `silent`. They may be cleared by entering

```
?- -flag.
```

The behaviour of these flags can be summarised as follows.

tracer
: When running in verbose mode, the parser describes which chart edges are being combined and which grammar rules are being invoked.

phrasal
: When storing signs in the chart, the system does not normally store their constituency information as a value of the `dtrs` feature. This results in a great improvement in efficiency, and if the information is needed, it can easily be reconstructed from the chart [PV91]. If the `phrasal` flag is set, then for each successful parse this information will be reconstructed.

explicit
: Setting the `explicit` flag will cause the system to explicitly store the `dtrs` information of a sign in the chart. This can sometimes be useful for debugging purposes.

silent
: By setting this flag, the user can eliminate all parser output. This is useful if the `parse` predicate is being called by some other process which also does all input and output.

After the chart parser has finished processing a sentence, the TreeTool module can be invoked to display the signs associated with each successful parse. This request for a graphic display of the parse is achieved with the goal

```
?- gparse.
```

TreeTool can also be used to examine all of the edges in the chart by issuing the command

```
?- gchart.
```

By default, the parser will consider a sentence to be grammatical if it can construct a phrasal sign that spans the entire input sentence. The user may place restrictions on phrasal signs for grammatical sentences by specifying a phrasal sign (corresponding to a "start symbol" in a traditional grammar) in the `.lex` file. For example, we may want the system only to consider *saturated* phrasal signs for successful parses. This could be achieved by the following entry which makes use of previously defined lexical types.

```
ideal @psign & @sat.
```

The format for the `ideal` command is the same as that for lexical entries.

$$\texttt{ideal}\ \ M_1\ \texttt{\&}\ M_2\ \texttt{\&}\ \ldots\ \texttt{\&}\ M_m\ \texttt{with}\ P_1\ \texttt{with}\ P_2\ \ldots\ \texttt{with}\ P_n\texttt{.} \tag{15}$$

There may be more than one `ideal` entry in a grammar.

HPSG-PL contains a facility for handling unknown words encountered during parsing. Normally, if the parser encounters a word that is not in the lexicon then the parse fails. However, the user can specify AVMs that are to be associated with unknown words. Explicit lexical entries for unknown words can be included in the lexicon. The format for these lexical entries is similar to that of normal lexical entries, except that no *word* is given, and the keyword `unknown_word` is used in place of `entry`. Consider the following example.

```
unknown_word @lsign with
   syn:loc:head:adjuncts = [_] with
   syn:loc:lex = plus.
```

When an unknown word is encountered, the system will use this lexical entry and continue with the parse. Again, there may be more than one `unknown_word` entry in a `.lex` file.

4.4 Debugging a Grammar

The system currently possesses a very primitive debugging system. For the most part, you must rely on TreeTool, the trace provided by the parser, and the debugging tools from the Prolog system that you are using. We have supplied several predicates that make it easier to debug a grammar.

look/nolook
: Calling the predicate `look(`N`)` where N is an integer will cause the system to call the Prolog `trace` predicate when edge N is being processed by the chart parser. The call `nolook(`N`)` will disable the tracing of edge N.

pedge
: The predicate `pedge(`N`)` where N is an integer will pretty print the AVM associated with edge N in the chart.

prule
: The predicate `prule(`N`)` where N is the *Name* of a rule, will pretty print the AVM associated with the rule.

psign

The predicate **psign**(N) where N is the *Name* of a lexical type, will pretty print the AVM associated with the lexical type.

pp

The call **pp**(X) where X is either an AVM, AVP, list or set will display the structure X in a human readable form.

4.5 Error Messages

If the lexical compiler fails in converting a lexical type, lexical entry, rule, etc. into its corresponding Prolog term, then an error message of the following form will be given.

`Expansion failure for` *Expression*

If a path equation contains an attribute name that has not been defined, then the system will display the following error message.

`Undefined feature in path:` *Path*

For example, issuing the following declaration with the undefined feature **sons**

```
example  type  @sign with sons = [].
```

will result in the following error messages.

```
Undefined feature in path: sons
Expansion failure for @sign with sons=[]
```

Finally, in the case where a path equation attempts to access a feature in an uninstantiated AVM, then the system displays the message

`Taking path of uninstantiated matrix:` *Path*

For example, the AV declarations given in Appendix A do not describe the structure of the **sem:cont** attribute. Thus, if a declaration contained reference to something like **sem:cont:pred**, the system would display an error message as shown below.

```
example  type  @sign with sem:cont:pred = walk.

Taking path of uninstantiated matrix: pred
Expansion failure for @sign with sem:cont:pred=walk
```

5 Summary

In closing, we should stress that the HPSG-PL system was not developed as a robust grammar development system for unification based grammar formalisms in general. Thus, it differs from formalisms like PATR-II [SUP+83], the Stuttgart Unification Formalism (STUF) [BKU88] and Comprehensive Unification Formalism (CUF) [ED90] which are intended to be general purpose *meta*-formalisms. HPSG-PL contains the bare necessities for the development and testing of HPSG grammars. It was not necessary to develop a system of *logical types* as has been done in some general purpose formalism. Instead all types, or operations that are based on types (e.g. equality tests) are based on structures instead of names. Additionally, the system does not currently possess facilities to allow the definition of arbitrary functions for use within AVMs—the user may only use the predefined functions `append`, `order`, `select`, `select1` and `combine`. Modifications to the definitions of these functions, or the definition of new functions, must be achieved by modifying the Prolog code.

Acknowledgements

Special thanks go out to Dan Fass for his comments and suggestions. The research presented in this paper is supported by a fellowship from the Advanced Systems Institute of British Columbia, and by the Natural Sciences and Engineering Research Council of Canada under Grant Nos. OGP0041910.

References

[BHP90] Sue Baker, Rob Hamm, and Fred Popowich. The TreeTool user's manual. Technical Report CMPT TR 90-09, Simon Fraser University, Burnaby, B.C., 1990.

[BKU88] Gosse Bouma, Ester Koenig, and Hans Uszkoreit. A flexible graph-unification formalism and its application to natural language processing. *IBM Journal of Research and Development*, 32:170–184, 1988.

[ED90] A. Eisele and J. Dörre. A comprehensive unification formalism. DYANA interal memo, Institut für maschinelle Sprachverarbeitung, University of Stuttgart, September 1990.

[EG84] Roger Evans and Gerald Gazdar. *The ProGram Manual.* School of Cognitive and Computing Sciences, University of Sussex, 1984.

[MCK+89] Marc Moens, Jo Calder, Ewan Klein, Mike Reape, and Henk Zeevat. Expressing generalizations in unification-based grammar formalisms. In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics.* Manchester, England, 1989.

[Mel88] Chris Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1):40–51, 1988.

[Pop85] Fred Popowich. Saumer: Sentence analysis using metarules. In *Proceedings of the 2nd Meeting of the European Chapter of the Association for Computational Linguistics*, pages 48–56. Geneva, Switzerland, March 1985.

[PS87] Carl Pollard and Ivan Sag. *Information-Based Syntax and Semantics, Volume 1: Fundamentals.* Centre for the Study of Language and Information, Stanford University, CA, 1987.

[PV90] Fred Popowich and Carl Vogel. Chart parsing head-driven phrase structure grammar. Technical Report CSS-IS TR 90-01, CMPT TR 90-01, Simon Fraser University, Burnaby, B.C., 1990.

[PV91] Fred Popowich and Carl Vogel. A logic based implementation of head-driven phrase structure grammar. In C.G. Brown and G. Koch, editors, *Natural Language Understanding and Logic Programming, III*, pages 227–246. Elsevier, North-Holland, 1991.

[SUP+83] Stuart Shieber, Hans Uszkoreit, Fernando Pereira, Jane Robinson, and M. Tyson. The formalism and implementation of PATR-II. In B. Grosz and M. Stickel, editors, *Research on Interactive Acquisition and Use of Knowledge*, pages 39–79. SRI International, Menlo Park, CA., 1983.

A Sample Grammar

The following sample grammar incorporates the three grammar rules introduced in [PS87], plus it includes two grammar rules for treating adjuncts. The grammar does not include a topicalization rule, nor does it currently include a version of the Binding Inheritance Principle. The Semantics Principle simply unifies the semantics of the head daughter with that of the complements and adjuncts. In the sample lexicon, semantic information for the individual lexical entries has not been provided.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% examples.typ
%
%    examples of type declarations for use with HPSG-PL system
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Semantics

sem         has  [!index, !cont].


% Syntax
```

```
agr      has  [!num, !per, !gen].
maj      has  [!n,!v].
head     has  [@maj, !min, !form, !inv, !aux, !case, @agr, !adjuncts].
loc      has  [@head, !subcat, !lex].
slash    has  [!syn, @sem].            % Avoid recursive type
inher    has  [@slash, !rel, !que].
to_bind  has  [@slash, !rel, !que].
non_loc  has  [@inher, @to_bind].
syn      has  [@loc, @non_loc].


% Define the domains for our set-valued features...

form     domain  [fin, bse, inf, prp, psp, pas].
case     domain  [nom, acc, gen].

% And use the following to make the grammar more concise.

fin      type    form set [fin].
bse      type    form set [bse].
inf      type    form set [inf].
prp      type    form set [prp].
psp      type    form set [psp].
pas      type    form set [pas].
part     type    form set [prp,psp].

nom      type    case set [nom].
obj      type    case set [acc].


%%%%% Lexical Types (the lexical hierarchy) %%%%%

% Different types of signs

sign     type [!dtrs, !phon, @syn, @sem].

lsign    type @sign with dtrs = [].

psign    type @sign with                          % Phrasal Sign
   dtrs = [!head_dtr, !comp_dtrs, !adj_dtrs, !filler_dtrs] with
   dtrs:head_dtr = @sign with
   dtrs:filler_dtrs = [].

sat      type @sign with                          % Saturated
   syn:loc:subcat = [].

unsat    type @sign with                          % Unsaturated
   syn:loc:subcat = [_|_].

no_adjn type @sign with                           % A sign with no adjuncts
```

```
   syn:loc:head:adjuncts = □.

% Category class definitions

det     type @sign & @sat & @no_adjn with
   syn:loc:head:maj:n = undef with
   syn:loc:head:maj:v = undef with
   syn:loc:head:min = d.

adj     type @sign & @sat & @no_adjn with
   syn:loc:head:maj:n = plus with
   syn:loc:head:maj:v = plus with
   syn:loc:head:min = undef.

noun    type @sign with
   syn:loc:head:form = norm with
   syn:loc:head:min = undef with
   syn:loc:head:maj:n = plus with
   syn:loc:head:maj:v = minus.

prep    type @sign & @unsat  & @no_adjn with
   syn:loc:head:min = undef with
   syn:loc:head:maj:n = minus with
   syn:loc:head:maj:v = minus.

verb    type @sign with
   syn:loc:head:min = undef with
   syn:loc:head:maj:n = minus with
   syn:loc:head:maj:v = plus.

adjnoun type @sign with
   syn:loc:head:min = undef with
   syn:loc:head:maj:n = plus.

prepverb type @sign with
   syn:loc:head:min = undef with
   syn:loc:head:maj:n = minus.


% Definitions for types appearing on SUBCAT lists

nonverbal type @sign & @unsat with
   syn:loc:head:maj:v = minus with
   syn:loc:lex = minus.

nominal   type @noun & @unsat with
   syn:loc:lex = minus.

np(Case)  type @noun & @sat with            % This is a parameterized lexical type.
   syn:loc:head:case = Case.                % "Case" is a variable.
```

```
xcomp(Form) type @verb with
   syn:loc:lex = minus with
   syn:loc:head:aux = minus with
   syn:loc:head:form = Form with
   syn:loc:subcat = [@np(@nom)].

% SUBCAT class definitions

intrans type @unsat with
   syn:loc:subcat = [@np(@nom)].

% Preposition class definitions (needed before we can continue with SUBCAT definitions)

prep(Form) type @prep with
   syn:loc:head:form = Form.

pp(Form) type @prep(Form) & @intrans with
   syn:loc:lex = minus.

% more SUBCAT information

trans   type @unsat with
   syn:loc:subcat = [@np(@obj),@np(@nom)].

ditrans type @unsat with
   syn:loc:subcat = [@np(@obj),@np(@obj),@np(@nom)].

/*
  In a version where we have a semantic attribute, the semantic index will be
  a parameter of the np and the xcomp... the following would need to reflect
  the appropriate coindexing.
*/

icontrol(Form) type @unsat with                    % intrans-control
   syn:loc:subcat = [@xcomp(Form),@np(@nom)].

tcontrol(Form) type @unsat with                    % trans-control
   syn:loc:subcat = [@xcomp(Form),@np(@obj),@np(@nom)].


% Noun class definitions

common(Adjs)   type @noun & @unsat with            % Common nouns
   syn:loc:subcat = [@det] with
   syn:loc:head:adjuncts = Adjs.

pn      type @np(_) & @no_adjn.                    % Proper Nouns
```

```
% Verb class definitions

mainv(Form) type @verb with
   syn:loc:head:adjuncts = [@pp(_)] with
   syn:loc:head:aux = minus with
   syn:loc:head:inv = minus with
   syn:loc:head:form = Form.

auxil\Phon) type @verb with
   syn:loc:head:adjuncts = [] with
   phon = [Phon] with
   syn:loc:lex = plus with
   syn:loc:head:aux = plus with
   syn:loc:head:form = @fin.


% Types for lexical entries

word(Phon) type @lsign with
   phon = [Phon] with
   syn:loc:lex = plus with
   syn:loc:head:inv = minus.


% Parameterized lexical types for use in dictionary

adj(Phon)        type @word(Phon) & @adj.
det(Phon)        type @word(Phon) & @det.
cn(Phon)         type @word(Phon) & @common([@adj,@nonverbal]).
cn(Phon,Adjs)    type @word(Phon) & @common(Adjs).
pn(Phon)         type @word(Phon) & @pn.
preposition(P)   type @word(P) & @trans & @prep(P).
auxv(Phon)       type @auxil(Phon) & @icontrol(@bse).
transv(Phon,Form)          type @word(Phon) & @trans & @mainv(Form).
intransv(Phon,Form)        type @word(Phon) & @intrans & @mainv(Form).
ditransv(Phon,Form)        type @word(Phon) & @ditrans & @mainv(Form).
icontrolv(Phon,Form,XForm) type @word(Phon) & @icontrol(XForm) & @mainv(Form).
tcontrolv(Phon,Form,XForm) type @word(Phon) & @tcontrol(XForm) & @mainv(Form).


%%%%%  Principles %%%%%

% Head Feature Principle

hfp type @psign with
   syn:loc:head = Sign with
   dtrs:head_dtr:syn:loc:head = Sign.

% Subcategorization Principle
```

```
subp type @psign with
   syn:loc:subcat = Rest with
   syn:loc:lex = minus with        % modification to SUBCAT principle
   dtrs:comp_dtrs = Comps with
   dtrs:head_dtr:syn:loc:subcat = append(Comps,Rest).

% Constituent Ordering Principle

cop type @psign with
   dtrs = Dtrs with
   phon = order(Dtrs).

% Semantics Principle

semp type @psign with
   dtrs:head_dtr:sem:index = S with
   dtrs:head_dtr:sem:cont = HSem with
   dtrs:comp_dtrs = Comps with
   dtrs:adj_dtrs = Adjs with
   sem:index = S with
   sem:cont = combine(HSem,Comps,Adjs).

principles type @hfp & @subp & @cop & @semp.


%%%%% Rules %%%%%

csign type @psign with dtrs:adj_dtrs = □.   % Complement sign
asign type @psign with dtrs:comp_dtrs = □.  % Adjunct Sign

% Nonlexical Head daughter combines with final complement

1       rule    @principles & @csign with
                dtrs:head_dtr:syn:loc:lex = minus with
                syn:loc:subcat = □ with
                dtrs:comp_dtrs = [_].

% Lexical Head daughter combines with all but final complement

2       rule    @principles & @csign with
                dtrs:head_dtr:syn:loc:lex =  plus with
                dtrs:head_dtr:syn:loc:head:inv = minus with
                syn:loc:subcat = [_].

% Lexical Head daughter combines with all complements

3       rule    @principles & @csign with
                dtrs:head_dtr:syn:loc:lex =  plus with
                dtrs:head_dtr:syn:loc:head:inv = plus with
```

```
                dtrs:adj_dtrs = [] with
                syn:loc:subcat = [].

% Unsaturated head combines with adjective or noun adjuncts (one at a time)

4       rule    @principles  & @asign & @unsat with
                dtrs:head_dtr:syn:loc:lex = minus with
                dtrs:head_dtr:syn:loc:head:adjuncts = A with
                dtrs:adj_dtrs = [@adjnoun] with
                dtrs:adj_dtrs = select(A).

% Saturated head combines with verb or prep adjuncts (one at a time)

5       rule    @principles  & @asign & @sat with
                dtrs:head_dtr:syn:loc:lex = minus with
                dtrs:head_dtr:syn:loc:head:adjuncts = A with
                dtrs:adj_dtrs = [@prepverb] with
                dtrs:adj_dtrs = select(A).

%%%
```

```
entry   @ditransv(give,@bse).
entry   @icontrolv(to,@inf,@bse).
entry   @icontrolv(seems,@fin,@inf).  % raising
entry   @icontrolv(seem,@bse,@inf).
entry   @icontrolv(tries,@fin,@inf).  % equi
entry   @icontrolv(try,@bse,@inf).
entry   @tcontrolv(persuades,@fin,@inf). % equi
entry   @tcontrolv(persuade,@bse,@inf).
entry   @tcontrolv(believes,@fin,@inf). % raising
entry   @tcontrolv(believe,@bse,@inf).
entry   @auxv(does).
```

B Terminal Session

The following terminal session was obtained using Quintus Prolog on a Sun SPARCstation 1.

```
| ?- [main].
% compiling file /tmp_mnt/home/dk2/ugg/HPSG-prolog/Distrib/main.pl
%  compiling file /tmp_mnt/home/dk2/ugg/HPSG-prolog/Distrib/lex.pl
%  lex.pl compiled in module user, 1.217 sec 4,492 bytes
%  compiling file /tmp_mnt/home/dk2/ugg/HPSG-prolog/Distrib/parser.pl
%  parser.pl compiled in module user, 1.184 sec 4,932 bytes
%  compiling file /tmp_mnt/home/dk2/ugg/HPSG-prolog/Distrib/constraints.pl
%  constraints.pl compiled in module user, 0.600 sec 1,920 bytes
%  compiling file /tmp_mnt/home/dk2/ugg/HPSG-prolog/Distrib/utils.pl
%  utils.pl compiled in module user, 1.133 sec 4,244 bytes
%  compiling file /tmp_mnt/home/dk2/ugg/HPSG-prolog/Distrib/io.pl
%  io.pl compiled in module user, 0.533 sec 1,972 bytes
%  compiling file /tmp_mnt/home/dk2/ugg/HPSG-prolog/Distrib/treetool.pl
%  treetool.pl compiled in module user, 0.950 sec 2,980 bytes
% main.pl compiled in module user, 6.000 sec 23,360 bytes

yes
| ?- gram(example).

yes
| ?- parse.
|: the cat eats the cookie on the table.

Parse Found at 3.8 secs.
[[index,_4017],[cont,_4023]]

Parse Found at 4.616 secs.
[[index,_4027],[cont,_4033]]

Edges Processed: 43
Done at 4.983 secs.
```

```
yes
| ?- +tracer.

yes
| ?- parse.
|: the cat walks.
initializing:  create inactive edge 1 ([the]), built from entry the.
initializing:  create inactive edge 2 ([cat]), built from entry cat.
initializing:  create inactive edge 3 ([walks]), built from entry walks.
predicter:  create inactive edge 4 ([walks]), built from edge(3) and rule(2).
predicter:  create active edge 5 ([walks]), built from edge(4) and rule(1).
predicter:  create inactive edge 6 ([cat]), built from edge(2) and rule(2).
predicter:  create active edge 7 ([cat]), built from edge(6) and rule(1).
predicter:  create active edge 8 ([cat]), built from edge(6) and rule(4).
predicter:  create active edge 9 ([cat]), built from edge(6) and rule(4).
completer:  create inactive edge 10 ([the,cat]), built from edge(1) and edge(7).
completer:  create inactive edge 11 ([the,cat,walks]), built from edge(10) and edge(5).
predicter:  create active edge 12 ([the,cat]), built from edge(10) and rule(5).

Parse Found at 0.667 secs.
Edge Number: 11
[[index,_3890],[cont,_3896]]
predicter:  create active edge 13 ([the,cat,walks]), built from edge(11) and rule(5).

Edges Processed: 13
Done at 0.884 secs.

yes
| ?- -tracer.

yes
| ?- +phrasal.

yes
| ?- parse.
|: the cookie cat walks.

Parse Found at 1.75 secs.
[[index,_3933],[cont,_3939]]

PSign Reconstructed at 2.466 secs.

Edges Processed: 24
Done at 2.816 secs.

yes
| ?- -phrasal.

yes
| ?- parse.
```

```
|: the cookie cat walks.

Parse Found at 1.75 secs.
[[index,_3933],[cont,_3939]]

Edges Processed: 24
Done at 2.1 secs.

yes
| ?- +explicit.

yes
| ?- parse.
|: the cookie cat walks.

Parse Found at 4.417 secs.
[[index,_9927],[cont,_4724]]

Edges Processed: 24
Done at 6.283 secs.

yes
| ?- -explicit.

yes
| ?- parse.
|: the small cookie cat eats the cat cookie on the cookie table.

Parse Found at 23.317 secs.
[[index,_4229],[cont,_4235]]

Parse Found at 26.317 secs.
[[index,_4243],[cont,_4249]]

Parse Found at 29.6 secs.
[[index,_4249],[cont,_4255]]

Parse Found at 32.984 secs.
[[index,_4263],[cont,_4269]]

Edges Processed: 128
Done at 36.017 secs.

yes
| ?- parse.
|: john believes mary to give the cat the cookie on the table.

Parse Found at 10.1 secs.
[[index,_4177],[cont,_4183]]
```

Centre for Systems Science

• Simon Fraser University established the **Centre for Systems Science** as a provincial centre for advanced study. The objective of CSS is twofold: coordinate and combine advanced systems research at SFU, and encourage application of research results within industry in order to further the technological development of British Columbia.

•CSS has assembled three research groups to streamline its efforts. The orientation of these groups is as follows: intelligent systems, microelectronics, and computer and communications systems. Because of shared research interests, this latter group merges with LCCR (**Laboratory for Computer and Communications Research**) for the publication of technical reports.

• CSS's three groups involve over 80 skilled faculty researchers principally from SFU's Schools of Engineering Science and Computing Science, but also draw substantially on Mathematics, Physics, Geography, Philosophy, Linguistics, Psychology, Communication, and Kinesiology.